

Semi-Contiguous Memory Allocation for Efficient Sequential-Access

Elias G. Khalaf and Ralph P. Tucci
Department of Mathematics and Computer Science
Loyola University New Orleans
6363 St. Charles Ave., Campus Box 35
New Orleans, LA 70118
E-mail: {ekhalaf, tucci}@loyno.edu

Abstract

Many algorithms have been devised and studied for dynamic contiguous memory allocation. In the absence of enough contiguous memory to satisfy a particular request, one approach is to allocate memory in blocks that are as close as possible to each other, by minimizing interfering blocks – blocks that belong to previously allocated requests. We devise an algorithm for such an allocation strategy and discuss its efficiency. This allocation strategy would improve access time for sequential files like audio and video files.

Keywords: memory, allocation, semi-contiguous, dynamic, storage.

1. Introduction

Have you ever wanted to watch a movie with a bunch of friends, bought the tickets and went to the movie theater, only to find out that there is not a group of seats where you and your friends can sit together to watch the movie? Sometimes asking a person or two to move over one or two seats can solve the problem, but what about the next group of friends who want to sit together, let alone the friction that may result from asking someone to move! Another solution would be to ask everyone in the theater to move over to one side or to the middle, amounting to compacting the theater and freeing larger blocks of seats.

This problem is not new to computer science and deals with dynamic contiguous memory allocation, for which many efficient algorithms have been devised and studied.

But suppose that you and your friends are willing to split, but ideally would like to be as close to each other as possible. How would such a seating arrangement be achieved, such that a minimal number of people not in your group sit in between?

This problem is, in fact, at the heart of memory allocation algorithms, especially with secondary storage devices like hard disk drives where contiguous allocation of files is ideal for the most efficient retrieval, as it minimizes seek and rotational latency times. However, when contiguous allocation is not possible, mainly due to the lack of a contiguous block large enough to satisfy a given allocation request, the next best thing would be to allocate the blocks to be as close to each other as possible. This results in minimal seek and latency times, and thus more efficient retrieval.

In the following sections we define the problem of dynamic semi-contiguous memory allocation, describe our system model and state the assumptions under which we are working, and then devise an algorithm that efficiently allocates storage following our strategy. We then show an example of memory allocation using the algorithm we describe. Finally, we make some

conclusions and point out some future directions of this work.

2. Motivation and Related Work

Over the last 40 years, the increase in the speed of processors and main memory has far outstripped that for disk access, with processor and main memory speeds increasing by about two orders or magnitude compared to one order of magnitude for disk [3]. The result is that disks are currently at least four orders of magnitude slower than main memory. This gap is expected to continue into the foreseeable future. Thus the performance of disk storage subsystem is of vital concern, and much research has gone into schemes for improving that performance [3].

The main problem is how to allocate space to files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: *contiguous*, *linked*, and *indexed* [2], with each method having its advantages and disadvantages.

Contiguous allocation (described in more detail in the next section) supports both sequential and direct file access. However, it suffers from the difficulty of finding extra space for a file, and from *external fragmentation*. Linked allocation, which solves all the problems of contiguous allocation, allocates space for files as a linked list of disk blocks, which may be scattered anywhere on the disk. In addition to pointer allocation overhead, one other disadvantage is the disk time needed to access all the scattered blocks comprising a file. Another critical limitation is that linked allocation can only be used effectively for sequential-access files since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. Indexed allocation solves this problem, but incurs greater pointer overhead, and still suffers from the fact that data blocks may be spread all over the disk volume. Some systems may use a combination of the three allocation strategies described above, depending on the type of file access and file size, among other factors.

In this paper, we limit our scope to a variant of contiguous allocation. The algorithm we propose is optimal when mostly sequential-access

is used, like retrieving or storing large audio, video or graphics files. We do not know of any algorithm that approaches the problem of memory allocation in exactly this way.

3. System Model and Assumptions

As mentioned in the previous section, we limit our scope to the contiguous allocation method of allocating disk space to files. In this section we describe the parameters that affect disk performance and the contiguous allocation method. We then describe the system model and the assumptions underlying this model and the proposed allocation algorithm.

Disk Performance Parameters

Without digging deep into technical details of disks, we need to define the major parameters affecting the performance of a disk in terms of data access. When operating, disks rotate at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track. This is known as *seek time*. The time it takes for the beginning of the sector to reach the head is known as *rotational delay*, or *rotational latency*. The sum of the seek time and the rotational delay equals the *access time*, which is the time it takes to get into position to read or write. Once in position, the time it takes to read or write, as the sector moves under the head, is the *transfer time* [3].

Contiguous Allocation

Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk [2]. With this ordering, assuming that only one job is accessing the disk, accessing block $b + 1$ after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next. Thus, seek time is minimized [2]. Contiguous allocation can be seen as a particular application of the general *dynamic storage-allocation* problem, which

involves how to satisfy a request of size n from a list of free holes.

In this paper we describe a variant of contiguous allocation, which we call *semi-contiguous* allocation as described in the next section. The main idea is that, in the absence of a chunk of memory large enough for contiguous allocation, we go for the next-best strategy, which is to allocate a series of blocks that are not all contiguous, but as close to being so as possible. We do this by minimizing the number of intervening blocks – blocks belonging to other files.

System Model

Our memory model consists of M sequential blocks with addresses 0 through $M-1$, where each block is either free (available for allocation), or occupied (allocated). Free blocks are allocated to files upon request, and occupied blocks are unallocated when a file is deleted or must be moved from one place to another in memory. Figure 1 is an example snapshot of our memory model ($M=8$) in which a shaded block represents an occupied block and a non-shaded block a free one.

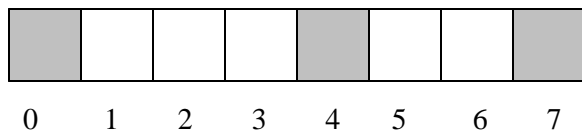


Figure 1. An 8-block memory snapshot

In the example above we have 8 memory blocks, three of which (0, 4, and 7) are occupied, and the rest free.

Definition A *hole* is a maximal set of contiguous free blocks. In the example above, the three blocks from 1 to 3 comprise a hole, and the two blocks 5 and 6 comprise another.

Definition A chunk of memory is *semi-contiguous* if it consists of two or more consecutive holes. In the example above, a memory allocation that occupies both holes results in a chunk of memory consisting of blocks 1-6 that is semi-contiguous.

Definition A *window* is a data structure consisting of

- L - leftmost address in a hole
- R - rightmost address in a hole
(This is not necessarily the same hole containing L)
- OCC - number of occupied blocks from L to R
- $FREE$ - number of free blocks from L to R

Note $FREE = R - L + 1 - OCC$

Example In the example in Figure 1, take the blocks from 1 to 6 inclusive. These blocks are described by a window as follows:

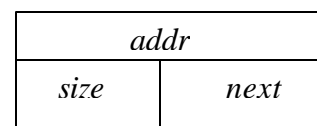
$$\begin{aligned}
 L &= 1 \\
 R &= 6 \\
 OCC &= 1 \\
 FREE &= 5
 \end{aligned}$$

Assumptions

Here are the assumptions we are working with for our algorithm:

- File sizes are known ahead of time. Our algorithm requires this information before it can determine which blocks to allocate.
- Current memory request cannot be fulfilled contiguously; otherwise, one of the standard contiguous allocation strategies can be used, like first fit, best fit, or worst fit.
- A linked-list data structure is available, which describes the list of holes available for allocation. Each node representing a hole consist of three values:
 - *addr* – The address in memory of the first block of the current hole.
 - *size* – The size of the current hole in blocks.
 - *next* – A pointer to the next node in the list.

We represent a node as follows:



A head pointer h to the first node in the list is available for initial access to the list, and the last node points to the special value nil . The linked list for the example in Figure 1 is shown in Figure 2 below.

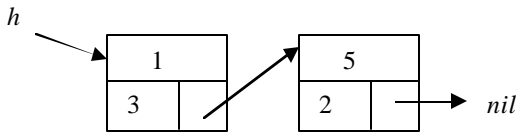


Figure 2. Linked-list representation of the example in Figure 1.

4. Semi-Contiguous Allocation Algorithm

Given a memory allocation request of size N blocks, we want to find a semi-contiguous chunk of memory large enough to satisfy the request, such that the number of intervening occupied blocks is minimal over the whole memory space. The algorithm works as follows:

1. Initially, the current window is set to encompass the first available hole, which is represented in the linked list by the first node. The number of free blocks is computed, and the number of intervening blocks is set to 0 (none yet since the first hole is contiguous).
2. As long as there are more nodes and the number of free blocks in the window is less than the request size (N), the window is expanded from the right by one hole at a time until the number of free blocks equals or exceeds N . Throughout this process the number of free blocks, as well as the number of intervening blocks, are computed. We also keep track of the address of the first block in the window with the smallest number of intervening blocks.
3. Once enough space is found for the request, the window is now shrunk by one hole from the left, and the process repeats from step 2.
4. Eventually, the window that satisfies the request size and has the smallest number of intervening blocks will be found.

5. If the number of intervening blocks goes down to 1 or 0, the algorithm halts as this is an optimal result.
6. Semi-contiguous allocation can now begin at the first hole in the best window found.

Note that the last hole in the best window may not be allocated completely as it could be larger than needed. In this case, a new smaller hole is produced, which can then be added to the free space list, thus avoiding external fragmentation.

Semi-Contiguous Allocation Algorithm

Input

N – the size of the file to be stored

A linked-list description of free memory

Output

$BESTL$ – the leftmost address of the first hole where semi-contiguous allocation of the request starts.

Local variables

L – leftmost address of current window

R – rightmost address of current window

$FREE$ – no. of free blocks from L to R inclusive

OCC – no. of occupied blocks in current window

$BESTL$ – leftmost address of best window so far

$BESTOCC$ – number of occupied blocks in the best window so far

$LPTR$ – a pointer to the leftmost node in window

$RPTR$ – a pointer to the rightmost node in window

BEGIN

/ Initialize */*

$LPTR = h$

$RPTR = h$

$L = LPTR.addr$

$R = L + LPTR.size - 1$

$OCC = 0$

$FREE = LPTR.size$

$BESTL = L$

$BESTOCC = +infinity$

/ Execute */*

While ($RPTR \rightarrow next \neq nil$ **and** $BESTOCC > 1$)

If ($FREE < N$) **then**

/ expand window by one hole to the right */*

$R = RPTR \rightarrow next.addr + RPTR \rightarrow next.size - 1$

```

OCC = OCC + (RPTR→next.addr -
              RPTR.addr - RPTR.size)
RPTR = RPTR→next
Else
  /* shrink window by one hole from the left */
  L = LPTR→next.addr
  OCC = OCC - (L - LPTR.addr - LPTR.size)
  LPTR = LPTR→next
End if
If (OCC < BESTOCC and FREE = N) then
  /* We have found a better fit. */
  BESTL = L
  BESTOCC = OCC
End if
FREE = R - L + 1 - OCC
End while
If (BESTOCC = +infinity) then
  Reject Request /* cannot satisfy request */
Else if (BESTOCC = 0) then
  Contiguous allocation starting at BESTL
Else
  Semi-contiguous allocation starting at BESTL
End if
END

```

Trace Example

We will now show how the algorithm works by tracing it using the snapshot of memory in Figure 3, for a memory allocation request of $N = 6$. For convenience, the memory snapshot has been divided into three lines of 8 blocks each, but it should be regarded as one contiguous set of 24 blocks.

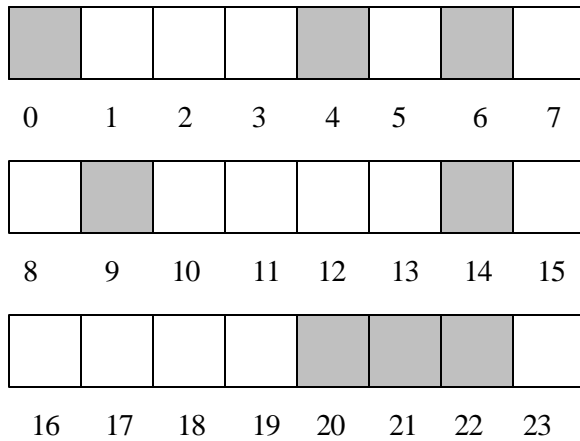


Figure 3. A 24-block memory snapshot

In the example in Figure 3 above we have 6 holes starting at blocks 1, 5, 7, 10, 15, and 23, respectively. The trace of the algorithm for this memory snapshot and request size $N=6$ is shown in Table 1.

CURRENT WINDOW				BEST WINDOW	
L	R	OCC	FREE	BESTL	BESTOCC
1	3	0	3	1	<i>+infinity</i>
1	5	1	4		
1	8	2	6	1	2
5	8	1	3		
5	13	2	7		
7	13	2	6		
10	13	0	4		
10	19	1	9	10	1

Table 1. A trace for $N=6$

Observations

The algorithm found that the best place to allocate such a request is starting at the hole at block 10, with *BESTOCC* of 1, i.e. only one intervening occupied block (block 14), which is really an optimal value.

The while loop in the algorithm described above stops when either we exhaust all nodes (holes) in the list, or when *BESTOCC* becomes 1, which is optimal given our assumptions. If *BESTOCC* is still *+infinity* even after the while loop terminates, then this means that the sum of all holes is not large enough for the request at hand, and the request is rejected. In our example, any request larger than 16 cannot be satisfied, and therefore will never have $FREE = N$, in which case *BESTOCC* will never get updated from its initial value of *+infinity*. We test for this condition in the if-statement immediately after the while loop.

If *BESTOCC* ever becomes 0 and $FREE = N$, then there is a hole that is large enough for contiguous allocation, which violates one of our assumptions. However, the algorithm will detect this condition in the Else-if-statement following the while loop, in which case it reduces to a first-fit contiguous allocation, which is generally superior to best-fit and worst-fit [1]. Again, this algorithm is supposed to be run when there is no hole large enough to accommodate a contiguous allocation of the request at hand. Otherwise, standard allocation techniques, like first fit, best fit, or worst fit, among others, may be used.

5. Algorithm Analysis

Given a memory space of M blocks, the most number of holes it could have is the ceiling of $M/2$. This happens when every other block is occupied. Our algorithm is linear in the number of holes since the sliding window moves over each hole twice (once from the right end, and once from the left end). Therefore, the worst case complexity is $O(M)$.

The only major space required in this algorithm is the singly linked-list of nodes representing holes. Each node requires three values (*addr*, *size*, and *next*). If each of the three values requires 4 bytes, then a total of 12 bytes is needed per node. As discussed previously, the maximum number of holes possible is roughly $M/2$, which takes place when every other hole is occupied. A typical block on a disk can hold 512 bytes of memory. Therefore, for every two blocks (1024 bytes), 12 bytes are needed, which is roughly 1.17% of the total memory available. The fewer the number of holes, the smaller this fraction gets. Although the free list can be too large to store in memory, there are techniques for storing parts of the list in memory at a time, like a push-down stack or FIFO queue [3].

6. Future Work

It seems possible to represent memory using a binary tree. This would make it quicker to find an optimal semi-contiguous chunk of memory, but it would require complex tree operations,

such as balancing in case of insertions and deletions. One other direction would be to use a bit vector representation of free space.

7. Conclusions

In this paper we have presented a dynamic memory allocation algorithm for finding a semi-contiguous chunk of memory. This would minimize the data transfer time from the disk, especially for sequential access files like audio and video files, which are time-sensitive. Our algorithm is designed to be executed when no hole available is large enough to accommodate a given request. However, should such a hole be encountered, the algorithm reduces to a first-fit strategy.

References

- [1] Knuth, Donald E., "The Art of Computer Programming, Volume 1: Fundamental Algorithms", 2nd edition, Addison Wesley, 1973.
- [2] Silberschatz, Galvin and Gagne, "Operating System Concepts", 7th edition, John Wiley & Sons, 2005.
- [3] Stallings, William, "Operating Systems, Internals and Design Principles", 5th edition, Pearson Prentice Hall, 2005.